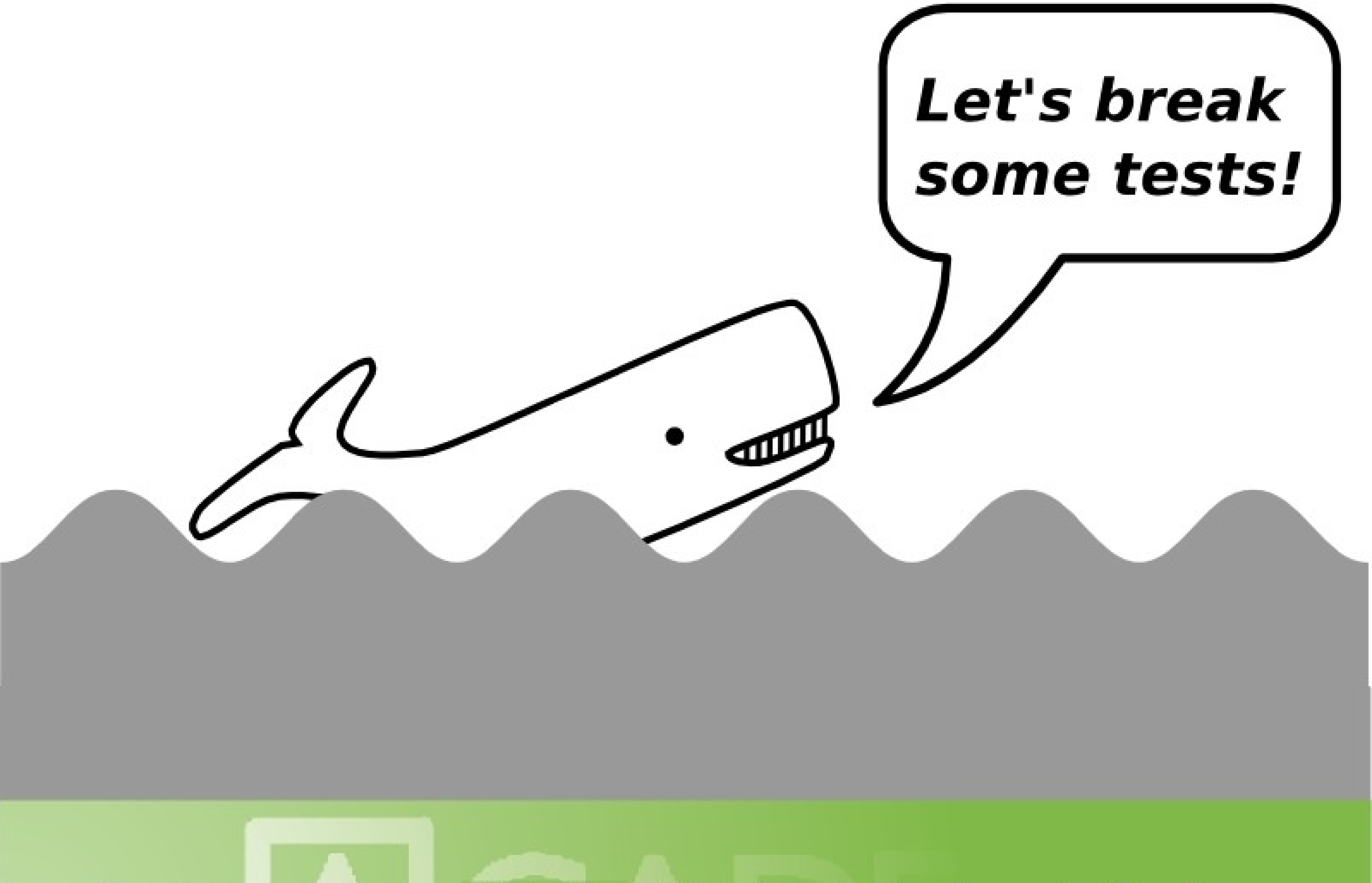
ACADEMIS

Python Testing Tutorial



featuring Melville's Moby Dick

Table of Contents

Python Testing Tutorial 1.1
Quotes 1.2
Warming Up 1.3

Exercises

Unit Tests 2.1
Fixtures 2.2
Parameterized Tests 2.3
Testing Command-Line Programs 2.4
Test Suites 2.5
Test Coverage 2.6
Testing New Features 2.7

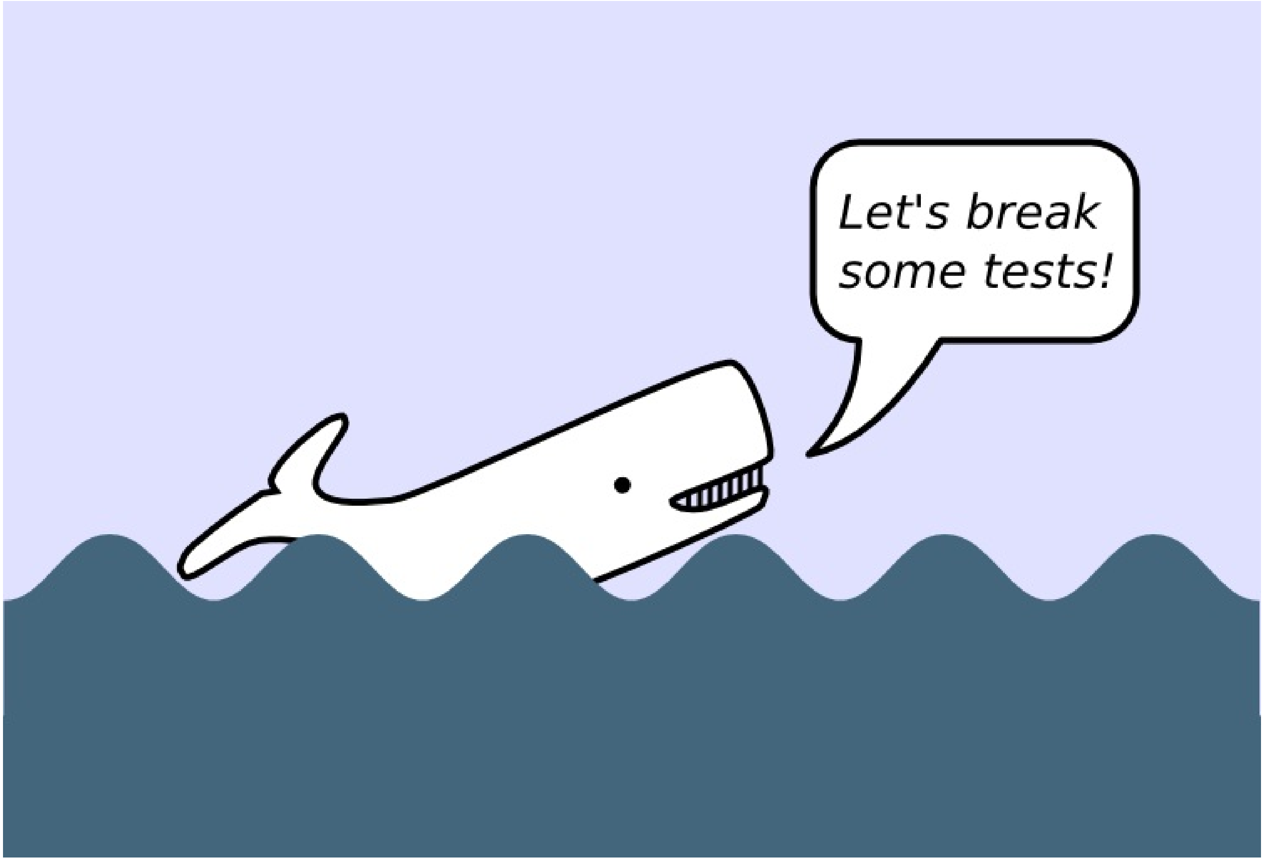
Instructions for Trainers

Theme: Counting Words in Moby Dick 3.1
Lesson Plan for a 45' tutorial 3.2
Lesson Plan for a 180' tutorial 3.3
Recap Puzzle 3.4

Python Testing Tutorial

Overview

This tutorial helps you to learn automated testing in Python 3 using the `py.test` framework.



Latest version of this book

- Sources for this tutorial: github.com/krother/python_testing_tutorial.
- PDF and EPUB versions: https://legacy.gitbook.com/book/krother/python-testing-tutorial/details

Copyright

Feedback and comments are welcome at: krother@academis.eu

© 2018 Magdalena & Kristian Rother

Released under the conditions of a Creative Commons Attribution License 4.0.

Contributors

Kristian Rother, Magdalena Rother, Daniel Szoska

Quotes

"Call me Ishmael"
Herman Melville, Moby Dick 1851

"UNTESTED == BROKEN"
Schlomo Shapiro, EuroPython 2014

"Code without tests is broken by design”
Jacob Kaplan-Moss

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"
Brian Kernighan, "The Elements of Programming Style", 2nd edition, chapter 2

"Pay attention to zeros. If there is a zero, someone will divide by it."
Cem Kaner

"If you don’t care about quality, you can't meet any other requirement”
Gerald M. Weinberg

"Testing shows the presence, not the absence of bugs."
Edsger W. Dijkstra

"... we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We're more of a testing, a quality software organization than we're a software organization."
Bill Gates (Information Week, May 2002)

Warming Up

How many words are in the following sentence?

```
The program works perfectly?
```

You will probably agree, that the sentence contains **four words**.

How many words are in the next sentence?

```
That #§&%$* program still doesn't work!\nI already
de-bugged it 3 times, and still numpy.array
keeps raising AttributeErrors. What should I do?
```

You may find the answer to this question less obvious. It depends on how precisely the special characters are interpreted.

What is automated testing good for?

Writing automated tests for your software helps you to:

- get clear on what you want the program to do.
- identify gaps in the requirements.
- prove the presence of bugs (**not their absence!**).
- help you during refactoring.

Unit Tests

Exercise 1: Test a Python function

The function **main()** in the module **word_counter.py** calculates the number of words in a text body.

For instance, the following sentence contains **three** words:

```
Call me Ishmael
```

Your task is to prove that the **TextCorpus** class calculates the number of words in the sentence correctly with **three**.

Run the example test in **test_unit_test.py** with

```
pytest test_unit_test.py
```

Exercise 2: Test proves if code is broken

The test in the module **test_failing_code.py** fails, because there is a bug in the function **word_counter.average_word_length()**. In the sentence

```
Call me Ishmael
```

The words are **four, two,** and **seven** characters long. This gives an average of:

```
>>> (4 + 2 + 7) / 3.0
4.333333333333333
```

Fix the code in `test_broken_code.py`, so that the test passes.

Exercise 3: Code proves if tests are broken

The test in the module **test_broken_test.py** fails, because there is a bug in the test file.

Your task is to fix the test, so that the test passes. Use the example in **test_broken_test.py**.

Exercise 4: Test border cases

High quality tests cover many different situations. The most common situations for the program **word_counter.py** include:

test case	**description**	**example input**	**expected output**
empty	input is valid, but empty	""	0
minimal	smallest reasonable input	"whale"	1
typical	representative input	"whale eats captain"	3
invalid	input is supposed to fail	777	*Exception raised*
maximum	largest reasonable input	*Melville's entire book*	*more than 200000*
sanity	program recycles its own output	*TextBody A created from another TextBody B*	*A equals B*
nasty	difficult example	"That #~&%* program still doesn't work!"	6

Your task is to make all tests in **test_border_cases.py** pass.

Fixtures

Exercise 1: A module for test data

Create a new module `conftest.py` with a string variable that contains a sentence with lots of special characters:

```
sample = """That #§&%$* program still doesn't work!
I already de-bugged it 3 times, and still numpy.array keeps raising AttributeErrors. W
hat should I do?"""
```

Create a function that returns a `mobydick.TextCorpus` object with the sample text above. Use the following as a header:

```
@pytest.fixture
def sample_corpus():
    ...
```

Exercise 2: Using the fixture

Now create a module `test_sample.py` with a function that uses the fixture:

```
def test_sample_text(sample_corpus):
    assert sample_corpus.n_words == 77
```

Execute the module with `pytest`. Note that you **do not** need to import `conftest`. Pytest does that automatically.

Exercise 3: Create more fixtures

Create fixtures for the two text corpora in the files `mobydick_full.txt` and `mobydick_summary.txt` as well.

Exercise 4: Fixtures from fixtures

Create a fixture in `conftest.py` that uses another fixture:

```
from mobydick import WordCounter

@pytest.fixture
def counter(mobydick_summary):
    return WordCounter(mobydick_summary)
```

Write a simple test that makes sure the fixture is not `None`

Parameterized Tests

Exercise 1: Sets of example data

You have a list of pairs (word, count) that apply to the text file `mobydick_summary.txt` :

```
PAIRS = [
         ('months', 1),
         ('whale', 5),
         ('captain', 4),
         ('white', 2),
         ('harpoon', 1),
         ('goldfish', 0)
]
```

We will create six tests from these samples.

Instead of creating six tests manually, we will use the **test parametrization in pytest**. Edit the file `test_parameterized.py` and add the following decorator to the test function:

```
@pytest.mark.parametrize('word, number', PAIRS)
```

Add two arguments `word` and `number` to the function header and remove the assignment below.

Run the test and make sure all six tests pass.

Exercise 2: Write another parameterized test

The function **get_top_words()** calculates the most frequent words in a text corpus. It should produce the following top five results for the book **mobydick_full.txt**:

position	word
1.	of
2.	the
3.	is
4.	sea
5.	ship

Write one parameterized test that checks these five positions.

Testing Command-Line Programs

Exercise 1: Test a command-line application

The program **word_counter.py** can be used from the command line to calculate the most frequent words with:

```
python word_counter.py mobydick_summary.txt
```

Command-line applications need to be tested as well. You find tests in **test_commandline.py**.

Your task is to make sure the command-line tests pass.

Exercise 2: Test command-line options

The program **word_counter.py** calculates most frequent words in a test file. It can be used from the command line to calculate the top five words:

```
python word_counter.py moby_dick_summary.txt 5
```

Your task is to develop a new test for the program.

Exercise 3: User Acceptance

The ultimate test for any software is whether your users are able to do what they need to get done.

Your task is to *manually* use the program **word_counter.py** to find out whether Melville used *'whale'* or *'captain'* more frequently in the full text of the book *"Moby Dick"*.

The User Acceptance test cannot be replaced by a machine.

Test Suites

Exercise 1: Test collection

Run all tests written so far by simply typing

```
pytest
```

Exercise 2: Options

Try some options of pytest:

```
pytest -v  # verbose output

pytest -lf # re-run failed tests

pytest -x  # stop on first failing test
```

Exercise 3: Fixing tests

Fix the tests in `test_suite.py`

Exercise 4: Test selection

Run only one test class

```
pytest test_suite.py::TestAverageWordLength
```

or a single test function:

```
pytest test_suite.py::TestAverageWordLength::test_average_words
```

Your task is to run only the function **test_word_counter.test_simple** from the test suite in **tests/**.

Test Coverage

For the next exercises, you need to install a small plugin:

```
pip install pytest-cov
```

Exercise 1: Calculate Test Coverage

Calculate the percentage of code covered by automatic tests:

```
pytest --cov
```

Exercise 2: Identify uncovered lines

Find out which lines are not covered by tests. Execute

```
coverage html
```

And open the resulting `htmlcov/index.html` in a web browser.

Exercise 3: Increase test coverage

Bring test coverage of **word_counter.py** to 100%.

Testing New Features

Exercise 1: Add new feature: special characters

Add a new feature to the **word_counter.py** program. The program should remove special characters from the text before counting words.

Your task is to prove that the new feature is working.

Exercise 2: Add new feature: ignore case

Add a new feature to the **word_counter.py** program. The program should ignore the case of words, e.g. *'captain'* and *'Captain'* should be counted as the same word.

Your task is to prove that the new feature is working.

Exercise 3: Add new feature: word separators

The program **word_counter.py** does separate words at spaces, but not tabulators. You need to change that.

The following sentence should also contain **four** words:

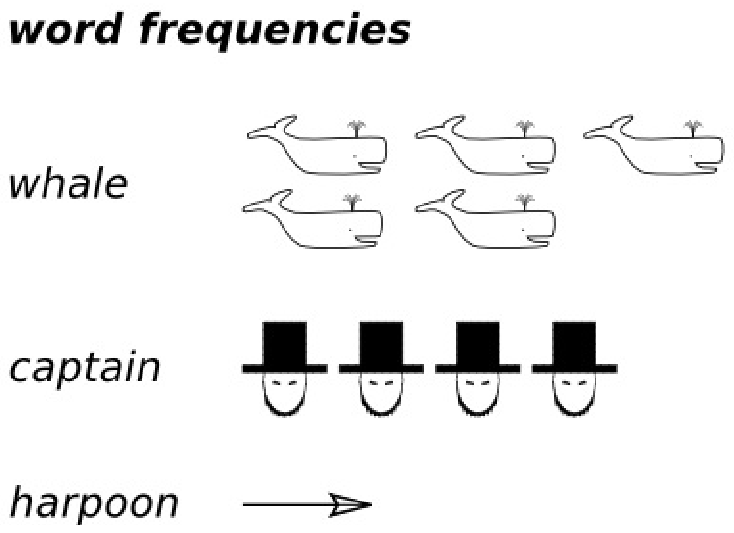
```
The\tprogram\tworks\tperfectly.
```

Your task is to add a test for this new situation and make it work.

Counting Words in Moby Dick

Moby Dick: Plot synopsis

Captain Ahab was vicious because Moby Dick, the white whale, had bitten off his leg. So the captain set sail for a hunt. For months he was searching the sea for the white whale. The captain finally attacked the whale with a harpoon. Unimpressed, the whale devoured captain, crew and ship. The whale won.



Video

[Moby Dick short synopsis on Youtube]

Course Objective

Herman Melville's book *"Moby Dick"* describes the epic fight between the captain of a whaling ship and a whale. In the book, the whale wins by eating most of the other characters.

But does he also win by being mentioned more often?

In this course, you have a program that analyzes the text of Melville's book.

You will test whether the program work correctly?

Why was this example selected?

Three main reasons:

- The implementation is simple enough for beginners.
- Counting words easily yields different results (because of upper/lower case, special characters etc). Therefore the program needs to be thoroughly tested.
- You can easily change the theme to another book from Project Gutenberg.

Lesson Plan for a 45' tutorial

Target audience

Programmers who have already written programs on their own but would like to learn about automated software testing.

Learning Objective

During the tutorial participants will implement automatic test functions that pass for the Moby Dick example. using the unittest module within 20'.

Lesson Plan

module	topic	time
warm-up	hello	1'
warm-up	question: How do you know that your code works?	4'
motivation	explain the benefit: You will be able to check in a few seconds that your program works.	1’
new content	overview of the code example	1’
new content	run the code example; collective analysis	15'
application	write code using the task description	20'
wrap-up	discuss pros and cons of testing	15'
wrap-up	point to materials	2'
wrap-up	goodbye	1'

Lesson plan for a 180' tutorial

I used a very similar lesson plan to conduct a training at EuroPython 2014. The audience consisted of about 60 Python programmers, including beginners and seasoned developers.

module	topic	time
warm-up	introduce the Moby Dick theme	5'
warm-up	icebreaker activity	5'
warm-up	announce training objectives	5'
part 1	**Writing automatic tests in Python**	45’
warm-up	methods in the unittest module	5’
new content	presentation: Unit Tests, Integration Tests, and Acceptance Tests	15’
application	challenges 1.1 - 1.5	20’
wrap-up	Q & A	5’
part 2	**Integration and Acceptance Tests (45')**	
warm-up	quiz on test strategies	10'
new content	presentation on Test-Driven-Development	10’
application	challenges 2.1 - 3.3	20'
wrap-up	Q & A	5’
break		10’
part 3	**Tests data and test suites (45')**	
warm-up	multiple choice questions	10'
new content	presentation on test suites	10’
application	exercises 4, 5, 6	20'
wrap-up	Q & A	5’
summary	**Benefits of testing (25')**	
transfer	group discussion on benefits of testing	20’
finishing	summary	4’
finishing	goodbye	1'

Recap Puzzle

The rows in the table got messed up! Match the test strategies with the correct descriptions.

test strategy	description
Unit Test	files and examples that help with testing
Acceptance Test	collection of tests for a software package
Mock	relative amount of code tested
Fixture	tests a single module, class or function
Test suite	prepare tests and clean up afterwards
Test data	replaces a complex object to make testing simpler
Test coverage	tests functionality from the users point of view

This exercise works better when each element from the table is printed on a paper card.